"Express Mail" Mailing Label No. **EL960828010US**

PATENT APPLICATION
ATTORNEY DOCKET NO. SUN04-0182-MEG

SELECTIVELY DEFERRING THE EXECUTION OF INSTRUCTIONS WITH UNRESOLVED DATA DEPENDENCIES AS THEY ARE ISSUED IN PROGRAM ORDER

Inventors: Shailender Chaudhry and Marc Tremblay

BACKGROUND

Field of the Invention

[0001] The present invention relates to techniques for improving the performance of computer systems. More specifically, the present invention relates to a method and an apparatus for speeding up program execution by selectively deferring execution of instructions with unresolved data-dependencies as they are issued for execution in program order.

Related Art

[0002] Advances in semiconductor fabrication technology have given rise to dramatic increases in microprocessor clock speeds. This increase in

microprocessor clock speeds has not been matched by a corresponding increase in memory access speeds. Hence, the disparity between microprocessor clock speeds and memory access speeds continues to grow, and is beginning to create significant performance problems. Execution profiles for fast microprocessor systems show that a large fraction of execution time is spent not within the microprocessor core, but within memory structures outside of the microprocessor core. This means that the microprocessor systems spend a large fraction of time waiting for memory references to complete instead of performing computational operations.

[0003] Efficient caching schemes can help reduce the number of memory accesses that are performed. However, when a memory reference, such as a load operation generates a cache miss, the subsequent access to level-two (L2) cache or memory can require dozens or hundreds of clock cycles to complete, during which time the processor is typically idle, performing no useful work.

[0004] A number of techniques are presently used (or have been proposed) to hide this cache-miss latency. Some processors support out-of-order execution, in which instructions are kept in an issue queue, and are issued "out-of-order" when operands become available. Unfortunately, existing out-of-order designs have a hardware complexity that grows quadratically with the size of the issue queue. Practically speaking, this constraint limits the number of entries in the issue queue to one or two hundred, which is not sufficient to hide memory latencies as processors continue to get faster. Moreover, constraints on the number of physical registers that are available for register renaming purposes during out-of-order execution also limits the effective size of the issue queue.

[0005] Some designers have proposed a scout-ahead execution mode, wherein instructions are speculatively executed to prefetch future loads, but wherein results are not committed to the architectural state of the processor. For

example, see U.S. Patent No. 6,415,356, entitled "Method and Apparatus for Using an Assist Processor to Pre-Fetch Data Values for a Primary Processor," by inventors Shailender Chaudhry and Marc Tremblay. This solution to the latency problem eliminates the complexity of the issue queue and the rename unit, and also achieves memory-level parallelism. However, it suffers from the disadvantage of having to re-compute any computational operations that are performed while in scout-ahead mode.

[0006] Hence, what is needed is a method and an apparatus for hiding memory latency without the above-described drawbacks of existing processor designs.

SUMMARY

[0007] One embodiment of the present invention provides a system that facilitates deferring execution of instructions with unresolved data dependencies as they are issued for execution in program order. During a normal execution mode, the system issues instructions for execution in program order. Upon encountering an unresolved data dependency during execution of an instruction, the system generates a checkpoint that can subsequently be used to return execution of the program to the point of the instruction. Next, the system executes subsequent instructions in an execute-ahead mode, wherein instructions that cannot be executed because of an unresolved data dependency are deferred, and wherein other non-deferred instructions are executed in program order.

[0008] In a variation on this embodiment, if the unresolved data dependency is resolved during execute-ahead mode, the system enters a deferred execution mode, wherein the system executes deferred instructions. If all deferred instructions are executed during this deferred execution mode, the system returns

to normal execution mode to resume normal program execution from the point where the execute-ahead mode left off.

[0009] In a further variation, executing deferred instructions in the deferred execution mode involves: issuing deferred instructions for execution in program order relative to the other deferred instructions (but out-of-order relative to other previously executed instructions); deferring execution of deferred instructions that still cannot be executed because of unresolved data dependencies; and executing other deferred instructions that able to be executed in program order.

[0010] In a further variation, if some deferred instructions are again deferred, the system returns to execute-ahead mode at the point where execute-ahead mode left off.

[0011] In a further variation, generating the checkpoint involves saving the precise architectural state of the processor to facilitate subsequent recovery from exceptions that arise during execute-ahead mode or deferred mode.

[0012] In a variation on this embodiment, the system keeps track of data dependencies while executing instructions to facilitate determining if an instruction is subject to an unresolved data dependency. Keeping track of data dependencies can involve maintaining state information for each register, wherein the state information indicates whether or not a value in the register depends on an unresolved data-dependency.

[0013] In a variation on this embodiment, the unresolved data dependency can include: a use of an operand that has not returned from a preceding load miss; a use of an operand that has not returned from a preceding translation lookaside buffer (TLB) miss; a use of an operand that has not returned from a preceding full or partial read-after-write (RAW) from store buffer operation; and a use of an

operand that depends on another operand that is subject to an unresolved data dependency.

[0014] In a variation on this embodiment, if the system encounters a non-data-dependent stall condition while executing in normal mode or execute-ahead mode, the system moves to a scout-ahead mode. In scout-ahead mode, instructions are speculatively executed to prefetch future loads, but results are not committed to the architectural state of the processor. When the launch point stall condition (the unresolved data dependency or the non-data dependent stall condition that originally caused the system to move out of normal execution mode) is finally resolved, the system uses the checkpoint to resume execution in normal mode from the launch point instruction (the instruction that originally encountered the launch point stall condition).

[0015] In a further variation, the non-data-dependent stall condition can include: a memory barrier operation; a load buffer full condition; and a store buffer full condition.

[0016] In a variation on this embodiment, the system stores deferred instructions in a deferred buffer that is organized as a first-in first-out buffer.

[0017] In a variation on this embodiment, the system issues instructions for execution in program order from an instruction buffer that is organized as a first-in first-out buffer.

[0018] Note that the present invention has a number of advantages over traditional out-of-order designs. First, the instruction buffers are simple first-in-first-out (FIFO) buffers, which unlike conventional out-of-order execution windows, do not require complex circuitry for content-addressable-memory (CAM) operations. Moreover, the FIFO buffers are common SRAM cells with a single read port and a single write port. Thus, their area grows linearly with their size, and hence permits storage of thousands of deferred instructions. In fact,

given the density advantages of SRAM, these thousands of instructions can be stored in the same area as is used by a traditional out-of-order issue queue, which may only store only 128 instruction. (Note that although the above-described embodiment of the present invention uses FIFO buffers to implement both instruction buffer 108 and deferred buffer 112, other non-FIFO structures, such as SRAM, can be used to implement instruction buffer 108 and deferred buffer 112.)

[0019] Second, only dependent instructions are retained in the deferred buffer, because independent data-ready instructions are executed at once. This saves chip resources. Note that in traditional out-of-order designs, data-ready instructions that have issued continue to consume space in the issue queue (or equivalent structure) until they are retired.

[0020] Moreover, renaming of architectural registers to physical registers is not necessary, and so the complexity of the rename unit can be dispensed with. This means that the size of the out-of-order window is not limited by the register file size. Hence, the present invention facilitates extending out-of-order execution to a potentially unlimited number of instructions.

BRIEF DESCRIPTION OF THE FIGURES

[0021] FIG. 1A illustrates a processor in accordance with an embodiment of the present invention.

[0022] FIG. 1B illustrates a register file in accordance with an embodiment of the present invention.

[0023] FIG. 2 presents a state diagram which includes the execute-ahead mode in accordance with an embodiment of the present invention.

DETAILED DESCRIPTION

[0024] The following description is presented to enable any person skilled in the art to make and use the invention, and is provided in the context of a particular application and its requirements. Various modifications to the disclosed embodiments will be readily apparent to those skilled in the art, and the general principles defined herein may be applied to other embodiments and applications without departing from the spirit and scope of the present invention. Thus, the present invention is not intended to be limited to the embodiments shown, but is to be accorded the widest scope consistent with the principles and features disclosed herein.

Processor

[0025] FIG. 1A illustrates the design of a processor 100 in accordance with an embodiment of the present invention. Processor 100 can generally include any type of processor, including, but not limited to, a microprocessor, a mainframe computer, a digital signal processor, a personal organizer, a device controller and a computational engine within an appliance. As is illustrated in FIG. 1, processor 100 includes instruction cache 102, fetch unit 104, decode unit 106, instruction buffer 108, deferred buffer 112, grouping logic 110, memory 113, arithmetic logic unit (ALU) 114, ALU 116, branch pipe 118 and floating point unit 120.

[0026] During operation, fetch unit 104 retrieves instructions to be executed from instruction cache 102, and feeds these instructions into decode unit 106. Decode unit 106 forwards the instructions to be executed into instruction buffer 108, which is organized as a FIFO buffer. Instruction buffer 108 feeds instructions in program order into grouping logic 110, which groups instructions together and sends them to execution units, including memory pipe 122 (for

accessing memory 124), ALU 114, ALU 116, branch pipe 118 (which resolves conditional branch computations), and floating point unit 120.

[0027] If an instruction cannot be executed due to an unresolved data dependency, such as an operand that has not returned from a load operation, the system defers execution of the instruction and moves the instruction into deferred buffer 112. Note that like instruction buffer 108, deferred buffer 112 is also organized as a FIFO buffer.

[0028] When the data dependency is eventually resolved, instructions from deferred buffer 112 are executed in program order with respect to other deferred instructions, but not with respect to other previously executed non-deferred instructions. This process is described in more detail below with reference to FIG. 2.

Keeping Track of Dependencies

[0029] The present invention keeps track of data dependencies in order to determine if an instruction is subject to an unresolved data dependency. In one embodiment of the present invention, this can involve maintaining state information for each register, which indicates whether or not a value in the register depends on an unresolved data dependency.

For example, FIG. 1B illustrates a register file 130 in accordance with an embodiment of the present invention. Each register in register file 130 is associated with a "not-there" bit, which keeps track of whether a valid operand value is contained in the register, or if the operand cannot be produced because of an unresolved data dependency. For example, if the register is waiting for an operand to return from a load operation, the corresponding not-there bit is set to indicate that the desired operand value is not present in the register. When a subsequent instruction references a source operand value that is marked as not-

there, and generates a result that is stored in a destination register, the system marks the destination register as not-there to indicate that the value in the destination register also depends on the unresolved data-dependency. This can be accomplished by marking the not-there bit of the destination register with the "OR" of the not-there bits for source registers of the instruction.

State Diagram

[0030] FIG. 2 presents a state diagram which includes the execute-ahead mode in accordance with an embodiment of the present invention. The system starts in normal execution mode 202, wherein instructions are executed in program order as they are issued from instruction buffer 108 (see FIG. 1).

[0031] Next, if an unresolved data dependency arises during execution of an instruction, the system moves to execute-ahead mode 204. An unresolved data dependency can include: a use of an operand that has not returned from a preceding load miss; a use of an operand that has not returned from a preceding translation lookaside buffer (TLB) miss; a use of an operand that has not returned from a preceding full or partial read-after-write (RAW) from store buffer operation; and a use of an operand that depends on another operand that is subject to an unresolved data dependency.

[0032] While moving to execute-ahead mode 204, the system performs a checkpointing operation to generate a checkpoint that can be used, if necessary, to return execution of the process to the point where the unresolved data dependency was encountered; this point is referred to as the "launch point." (Note that generating the checkpoint can involve saving the precise architectural state of the processor to facilitate subsequent recovery from exceptions that arise during execute-ahead mode or deferred mode.) The system also "defers" execution of

the instruction that encountered the unresolved data dependency, and stores the deferred instruction in deferred buffer 112.

[0033] Within execute-ahead mode 204, the system continues to execute instructions in program order as they are received from instruction buffer 108, and any instructions that cannot execute because of an unresolved data dependency are stored in deferred buffer 112.

[0034] When the system is in execute-ahead mode 204, if an unresolved data dependency is finally resolved, the system moves into deferred execution mode 206, wherein instructions are executed in program order from deferred buffer 112. During deferred execution mode 206, the system attempts to execute deferred instructions from deferred buffer 112. Note that the system attempts to execute these instructions in program order with respect to other deferred instructions in deferred buffer 112, but not with respect to other previously executed non-deferred instructions (and not with respect to deferred instructions executed in previous passes through deferred buffer 112). During this process, the system defers execution of deferred instructions that still cannot be executed because of unresolved data dependencies and places these again-deferred instruction back into deferred buffer 112. The system executes the other instruction that *can* be executed in program order with respect to each other.

[0035] After the system completes a pass through deferred buffer 112, if deferred buffer 112 is empty, the system moves back into normal execution mode 202. This may involve committing changes made during execute-ahead mode 204 and deferred execution mode 206 to the architectural state of the processor, if such changes have not been already committed. It can also involve throwing away the checkpoint generated when the system moved into execute-ahead mode 204.

[0036] On the other hand, if deferred buffer 112 is not empty after the system completes a pass through deferred buffer 112, the system returns to

execute ahead mode to execute instructions from instruction buffer 108 from the point where the execute-ahead mode 204 left off.

[0037] If a non-data dependent stall condition arises while the system is in normal execution mode 202 or in execute-ahead mode 204, the system moves into scout mode 208. (This non-data-dependent stall condition can include: a memory barrier operation; a load buffer full condition; a store buffer full condition, or a deferred buffer full condition.) In scout mode 208, instructions are speculatively executed to prefetch future loads, but results are not committed to the architectural state of the processor.

[0038] Scout mode is described in more detail in U.S. Patent No. 6,415,356, entitled "Method and Apparatus for Using an Assist Processor to Pre-Fetch Data Values for a Primary Processor," by inventors Shailender Chaudhry and Marc Tremblay. It is also described in U. S. Provisional Application No. 60/436,539, entitled, "Generating Prefetches by Speculatively Executing Code Through Hardware Scout Threading," by inventors Shailender Chaudhry and Marc Tremblay (filed 24 December 2002). It is additionally described in U. S. Provisional Application No. 60/436,492, entitled, "Performing Hardware Scout Threading in a System that Supports Simultaneous Multithreading," by inventors Shailender Chaudhry and Marc Tremblay (filed 24 December 2002). The above listed references are hereby incorporated by reference herein to provide details on how scout mode operates.

[0039] Unfortunately, computational operations performed during scout-ahead mode need to be recomputed again, which can require a large amount of computational work.

[0040] When the original "launch point" stall condition is finally resolved, the system moves back into normal mode 202, and, in doing so, uses the

previously generated checkpoint to resume execution from the launch point instruction (the instruction that initially encountered the stall condition).

[0041] Note that the launch point stall condition is the stall condition that originally caused the system to move out of normal execution mode 202. For example, the launch point stall condition can be the data-dependent stall condition that caused the system to move from normal execution mode 202 to execute-ahead mode 204, before moving to scout mode 208. Alternatively, the launch point stall condition can be the non-data-dependent stall condition that caused the system to move directly from normal execution mode 202 to scout mode 208.

[0042] The foregoing descriptions of embodiments of the present invention have been presented for purposes of illustration and description only. They are not intended to be exhaustive or to limit the present invention to the forms disclosed. Accordingly, many modifications and variations will be apparent to practitioners skilled in the art. Additionally, the above disclosure is not intended to limit the present invention. The scope of the present invention is defined by the appended claims.